Coding Quiz

- Do not open the quiz until you are directed to do so.
- Read **all** the instructions first.
- The quiz contains 8 pages, including this one.
- The quiz contains two multi-part problems. You have 110 minutes to earn 100 points.
- Each problem requires submitting one or more .java files. Each file should be submitted on IVLE before the end of the session.
- Document your code well. For each algorithm, explain your solution in comments. For each function/method, explain what it is supposed to do and how it does it. At least 10% of the points will be related to your documentation.
- Write your code clearly using good Java style. Points will be deducted for badly written code.
- Be sure to test your code for correctness. A correct, working solution will always get significant credit.
- The quiz is open book. You may use any notes or textbooks. You may also use any documentation available on the Oracle Java website, i.e., `http://download.oracle.com/javase/1.4.2/docs`. You may *not* use any other sites on the internet, your mobile phone, or any other electronic device. We may be monitoring the outgoing network connections to detect cheating.
- Read through all the problems before starting. Read the problems carefully.
- Do not spend too much time on any one problem.
- Good luck!

Problem #	**Name**	**Possible Points**	**Achieved Points**
1	Counting Inversions	30	
2	FriendTracker	70	
Total:			

A Note on Cheating

- Students will be taking the coding quiz on Wednesday, March 2 and Thursday, March 3. The Coding Quiz officially ends on Thursday, March 3 at 7pm.
- **Any discussion of the quiz with other students prior to 7pm on March 3 will be considered cheating.** For example, it is considered cheating if a friend asks, "How hard was the quiz?" and you respond, "Not so bad!" You may not discuss any aspect of the quiz, whether it is easy or hard, whether you did well or badly, etc. And you certainly may not discuss the problems on the quiz. Even if your friend has already completed the quiz, you may not discuss the quiz with them. There is to be no discussion of the quiz until 7pm on Thursday, March 3. The rules are clearly defined so that there will be no question as to what is and is not allowed.
- It is easy to cheat and consider it a minor offense. Don't. There are more important things in life than getting a slightly higher or slightly lower score on a quiz, and being an honest and upright individual is one of those. Don't be a cheater.
- There may be small differences between different versions of the quiz, and if you simply copy answers from a different version, your cheating will be clear.
- You may not leave the room with a copy of the quiz, and you may not e-mail a copy of the quiz to anyone. After the quiz ends on Thursday, you will be provided with a copy for future reference.
- Any cheating that is detected or reported will be treated harshly to the most severe extent allowed by NUS, including a zero for the entire module (not just the quiz), and referral for potential further disciplinary action (which can lead to expulsion from NUS).

Problem 1. Inversions [30 points]

In this problem, we will examine inversions in a sequence of integers. Given a sequence of integers $\{a_1, a_2, \ldots, a_n\}$, an **inversion** is a pair (a_j, a_k) where $j < k$ but $a_j > a_k$. For example, in the sequence:

$$S_1 = \{1 \;\; 2 \;\; 5 \;\; 3 \;\; 10\}$$

there is one inversion: the pair $(5, 3)$. Given a sequence of numbers, your program should count the number of inversions. As another example, consider the sequence:

$$S_2 = \{1 \;\; 2 \;\; 10 \;\; 5 \;\; 3\}$$

This sequence contains three inversions: $(10, 5)$, $(10, 3)$, and $(5, 3)$. (Notice that by definition, an empty list or a list containing only one element contains no inversions.)

For this problem, submit a single java file *Inversions.java* which contains a single class `Inversions` that implements the static methods described below. For this problem, you may use any functionality available in the Java libraries (e.g., java.util.ArrayList). For all parts of this problem, you may assume that all the integers in the input lists are distinct.

(a) In the `Inversions` class, implement a static method:

```
static int countInversions(ArrayList<Integer> intArray)
```

The `countInversions` method should return the number of inversions in the specified array. For example, `countInversions(`S_1`)` should return 1, while `countInversions(`S_2`)` should return 3 (where S_1 and S_2 are defined as in the examples above).

Your algorithm should run in time $O(n^2)$. For extra credit, develop a version that runs in time $O(n \log n)$ time. (Be sure to complete all the questions on the quiz before working on an extra credit solution.)

(b) In the `Inversions` class, implement a static method:

```
static ArrayList<Integer>
createInversions(ArrayList<Integer> intArray, int numInversions)
```

The static `createInversions` method takes as input a sorted list of integers and an integer `numInversions` $< n$. It should return a permutation of the original list containing exactly `numInversions` inversions. That is:

- The output list is a permutation of the input list:
  for every set S, if S' = `createInversions`(S, i), then $sort(S) == sort(S')$.
- The output has exactly `numInversions` inversions:
  for every set S, if S' = `createInversions`(S, i), then `countInversions`$(S) == i$.

Note that while a list may have up to $n(n-1)/2$ inversions, for this problem, the `createInversions` method will never be required to create more than $n - 1$ inversions.

Problem 2. Keeping Track of Your Friends [70 points]

In this problem you will implement a data structure to support a simple facebook-style application. Your data structure will keep track of a set of *users*, each of whom has a set of *friends*. It will support queries regarding the number of friends two users have in common, and it will determine which is the most popular user. Read the entire problem first before beginning.

(a) (Sets of Friends.) First, you will implement a *Set* abstract data type for storing the sets of friends. Your solution should consist of a class called `FriendSet` (and submitted in a file *FriendSet.java*) and must implement the `ISet` interface, provided with this quiz. Each element in the set is a `String`, i.e., the name of a friend. Your class should contain two constructions:

1. The first constructor should take no parameters and should create an empty set.
2. The second constructor should take one parameter, another FriendSet, and create a copy of that set.

The `ISet` interface consists of the following methods:

- `add(String name)`: adds the specified string to the set. It returns *true* if the add succeeds, and *false* if the add fails. The add will fail if the name is an empty string, or if the name already exists in the set. If the set has already reached its maximum size (see below), then you should throw an exception.
- `intersection(ISet otherSet)`: returns a new set that is the intersection of the two sets.
- `size()`: returns the size of the set.
- `enumerateSet()`: returns an array of strings in the set.

See Figure 1 for an example execution. Where there is ambiguity or the specification is unclear, specify in comments the decisions and assumptions you are making in your implementation. Note the following assumptions/restrictions:

- You may assume that no individual has more than 100 friends. That is, the set will never contain more than 100 elements.
- At a minimum, the instersection operation should run in $O(n^2)$ time, and every other operation should run in $O(n)$ time. More efficient solutions are possible and will receive more credit. Implement the most efficient solution you can in the (limited) time provided. Explain (in comments) the trade-off in performance you have selected. Remember that a working solution that is less efficient will generally get more credit than a broken solution that is potentially more efficient.
- You may *not* use any of the Java collection classes. That is, you may not use any of the classes available in java.util.* for this part of the problem.
- Submit one file *FriendSet.java* for this problem. (If you need more than this one file, submit the others as well.)

```
ISet setA = new FriendSet();
boolean added = false;
added = setA.add("Joe"); // Returns true
added = setA.add("Mary"); // Returns true
added = setA.add("Sue"); // Returns true
added = setA.add("Joe"); // Returns false
int setSize = setA.size(); // Returns 3
String[] setList = setA.enumerateSet(); // Returns an array containing: [Joe, Mary, Sue]

ISet setB = new FriendSet(setA); // Creates a copy of setA
added = setB.add("Joe"); // Returns false
added = setB.add("Leo"); // Returns true
ISet setC = setA.intersection(setB); // Returns setC containing [Joe, Mary, Sue]
setSize = setC.size(); // returns 3

ISet setD = new FriendSet();
String John = "John";
for (int i=1; i<=100; i++){
    setD.add(John + Integer.toString(i, 10)); // Returns true
}
setSize = setD.size(); // returns 100
added = setD.add("John101"); // Throws an exception!
```

Figure 1: Example use of the `ISet` interface and the `FriendSet` class.

(b) (Maintaining the User Database.) Second, you will implement the main database which keeps track of the users in your system. Your solution should consist of a class called `UserDatabase` (and submitted in a file *UserDatabase.java*) and must implement the `IUserDB` interface, provided with this quiz. Your class should contain a single constructor which takes no parameters and creates an empty user database. The interface contains the following methods:

- `addUser(String name)`: adds a user to the database. It returns *true* if the add succeeds, and *false* if the add fails. The add will fail if the name is an empty string, or if the name already exists in the database. If the database has already reached its maximum size (see below), then you may throw an exception. (If you choose not to throw an exception, then all your code must continue to function correctly despite exceeding the limit.)
- `addFriend(String userOne, String userTwo)`: adds `userOne` into `userTwo`'s set of friends, and adds `userTwo` into `userOne`'s set of friends. If either user does not exist, then this routine should throw an exception. If the users are already friends, then it succeeds without error.
- `findFriendsInCommon(String userOne, String userTwo)`: returns an array containing the set of users who are friends of both users. It either user does not exist, then this routine should throw an exception. If there are no users in common, then it should return an empty array.
- `findMostPopularUser()`: returns the most popular user. The most popular user is the one who has the most friends. If there is a tie, then the routine may return any user that has a maximum number of friends. If there are no users in the database, then it should throw an exception.

Note the following assumptions/restrictions:

- You may assume that the database contains no more than 500 users.
- You may want to implement a separate class (say, *User*, submitted in *User.java*) to hold the information associated with each user (i.e., their name and friend set).
- For each user in the datbase, use an object of type `ISet` to store the friends of that user. Note, however, that your code will be tested using a *reference implementation* of the *ISet* interface (not the `FriendSet` implementation of `ISet` that you submit in the first part). Thus you may not add any special functionality to the `FriendSet` implementation, nor may you modify the interface or make any assumptions regarding the implementation of the set. (In addition, even if your implementation from the first part is not correct, you can get full credit on this part if your code works with our reference implementation of `ISet`.)
- You *may* use the Java collection classes for this part. In fact, you are required to store your user data in an *ArrayList* or a *HashMap*.
- At a minimum, adding users and adding friends should run in $O(n)$ time, while finding friends in common and finding the most popular user may take $O(n^2)$ (where the running times depend on the implementation of `ISet`). Implement the most efficient solution you

can in the (limited) time provided. However, remember that a working solution that is less efficient will generally get more credit than a broken solution that is potentially more efficient.

- Make sure your database responds correctly when it is empty (i.e., when there are no users).
- Submit the file *UserDatabase.java* and (if needed) the file *User.java* for this problem. (If you need more than these two files, submit the others as well.)

```
IUserDB userDB = new UserDatabase();
boolean added = false;
added = userDB.addUser("Alice"); // Returns true
added = userDB.addUser("Bob"); // Returns true
added = userDB.addUser("Charlie"); // Returns true
added = userDB.addUser("Diana"); // Returns true
added = userDB.addUser("Eli"); // Returns true
added = userDB.addUser("Joe"); // Returns true

userDB.addFriend("Alice", "Bob");
userDB.addFriend("Bob", "Charlie");

String[] commonFriends = userDB.findFriendsInCommon("Alice", "Charlie");
// Returns an array containing [Bob].

String[] commonFriends = userDB.findFriendsInCommon("Alice", "Alice");
// Returns an array containing [Bob].

userDB.addFriend("Alice", "Jim"); // Throws an exception: Jim is not a user!

userDB.addFriend("Charlie", "Alice");
userDB.addFriend("Charlie", "Bob"); // Already friends, but that's ok.
userDB.addFriend("Charlie", "Diana");
userDB.addFriend("Charlie", "Eli");
userDB.addFriend("Charlie", "Joe");

// Charlie's popularity is 5
String popular = userDB.findMostPopularUser(); // Charlie is the most popular.

userDB.addUser("Kate");
userDB.addUser("Larry");
userDB.addFriend("Bob", "Kate");
userDB.addFriend("Bob", "Larry");
userDB.addFriend("Bob", "Eli");

// Charlie's popularity is still 5
// Bob's popularity is also 5
popular = userDB.findMostPopularUser(); // May return Bob or Charlie.
```

Figure 2: Example use of the `IUserDB` interface and the `UserDatabase` class.